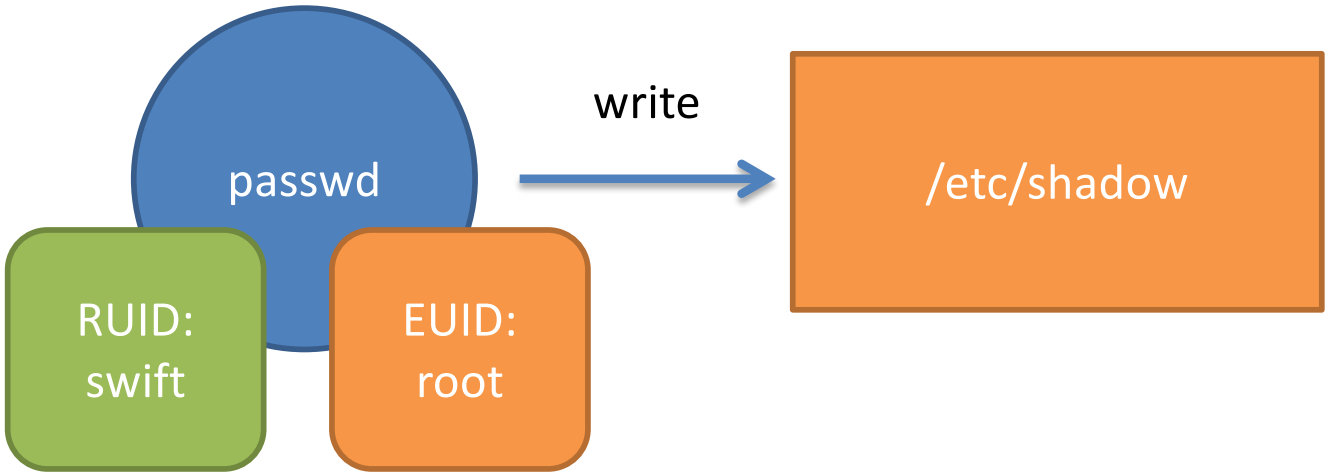
X86 Review
Process Layout, ISA, etc.

CS642:
Computer Security

Mike Swift

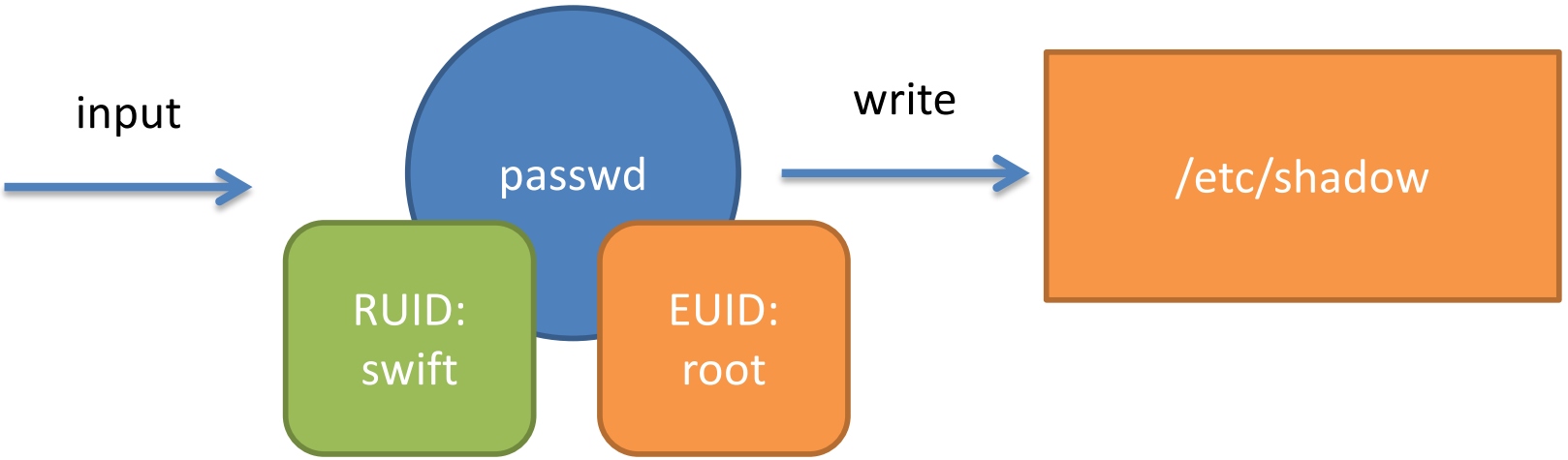
From Last Week

- ACL-based permissions (UNIX style)
  - Read, Write, eXecute can be restricted on users and groups
  - Processes (usually) run with the permissions of the invoking user
- Example:



Processes are the front line of system security

- Control a process and you get the privileges of its UID
- So how do you control a process?
  - Send specially formed input to process



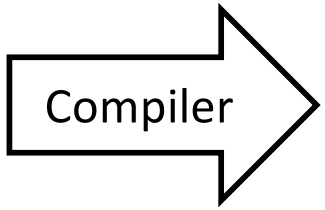
Roadmap

- Today
  - Enough x86 to understand (some) process vulnerabilities
- Next Time
  - How such attacks occur

Why do we need to look at assembly?

"WYSINWYX: What you see is not what you eXecute"
[Balakrishnan and Reps TOPLAS 2010]

```
memset(password, 'n0', len);
free(password);
```

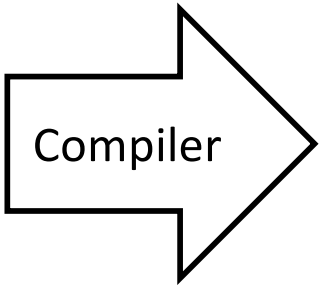



```
pushl 8(%ebp)
call free
```

Vulnerabilities exploited in this form

We understand code in this form

```
int foo(){
        int a = 0;
        return a + 7;
}
```

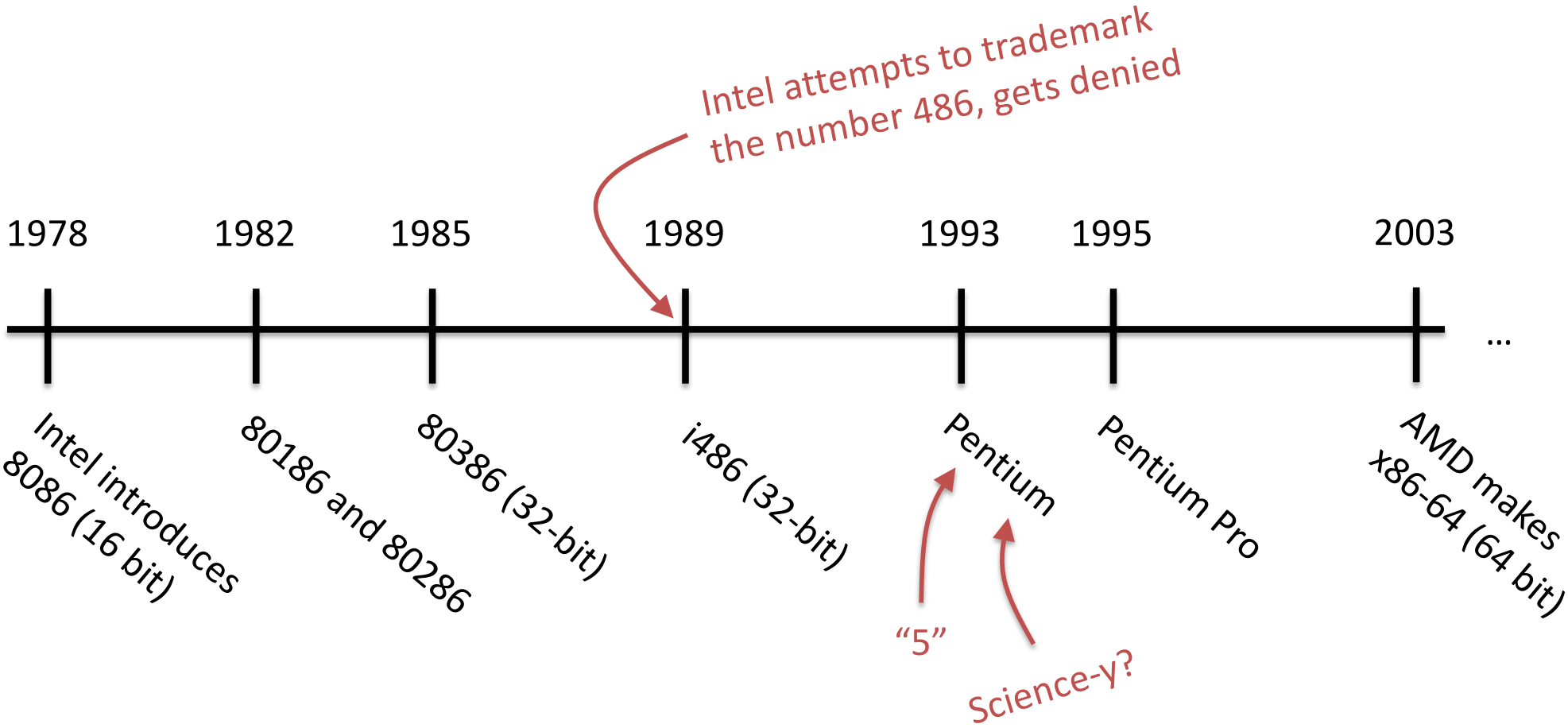



```
pushl %ebp
movl  %esp, %ebp
subl  $16, %esp
movl  $0, -4(%ebp)
movl  -4(%ebp), %eax
addl  $7, %eax
leave
ret
```

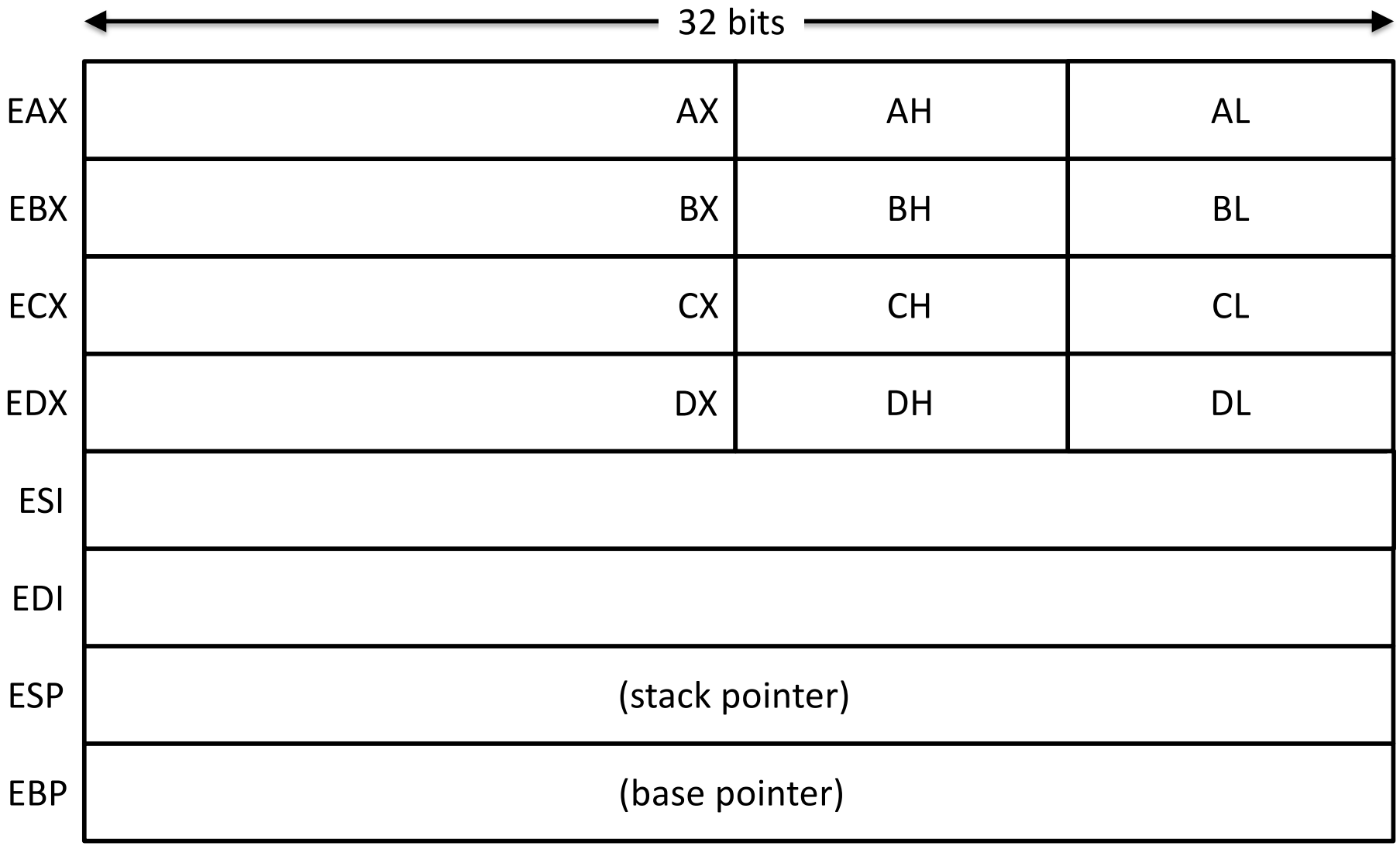
x86: Popular but crazy

- CISC (complex instruction set computing)
  - Over 100 distinct opcodes in the set
- Register poor
  - Only 8 registers of 32-bits, only 6 are general-purpose
- Variable-length instructions
- Built of many backwards-compatible revisions
  - Many security problems preventable… in hindsight

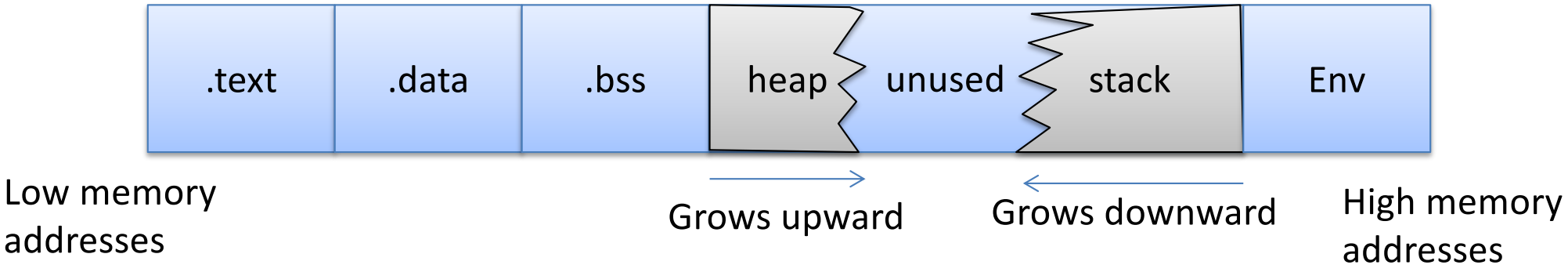
A Little History



Registers


32 bits
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)

Process memory layout



.text
- Machine code of executable

.data
- Global initialized variables

.bss
- Below Stack Section

global uninitialized variables

heap
- Dynamic variables

stack
- Local variables
- Function call data

Env
- Environment variables
- Program arguments

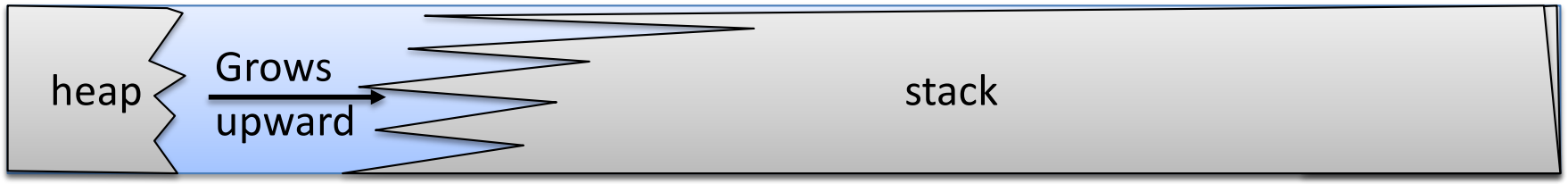
Heap and Stack Design

Low memory addresses

High memory addresses

- Allow for more efficient use of finite free memory
  - Growing in opposite directions allows extra flexibility at runtime
- Stack
  - Local variables, function bookkeeping
- Heap
  - Dynamic memory

Heap and Stack Design

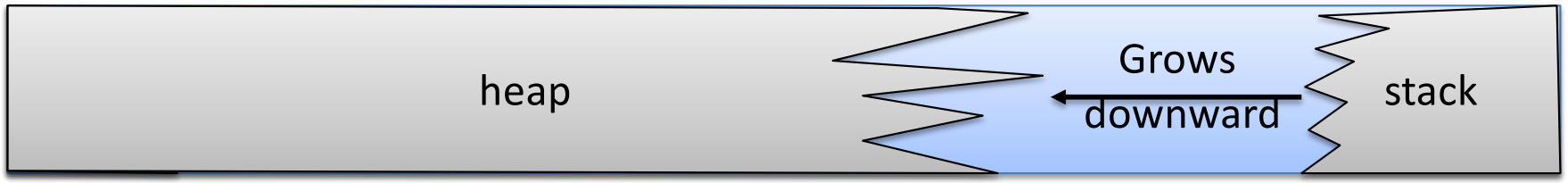


Low memory addresses

High memory addresses

- Allow for more efficient use of finite free memory
  - Growing in opposite directions allows extra flexibility at runtime
- Stack
  - Local variables, function bookkeeping
- Heap
  - Dynamic memory

Heap and Stack Design



Low memory addresses

High memory addresses

- Allow for more efficient use of finite free memory
  - Growing in opposite directions allows extra flexibility at runtime
- Stack
  - Local variables, function bookkeeping
- Heap
  - Dynamic memory

Heap and Stack use: Example

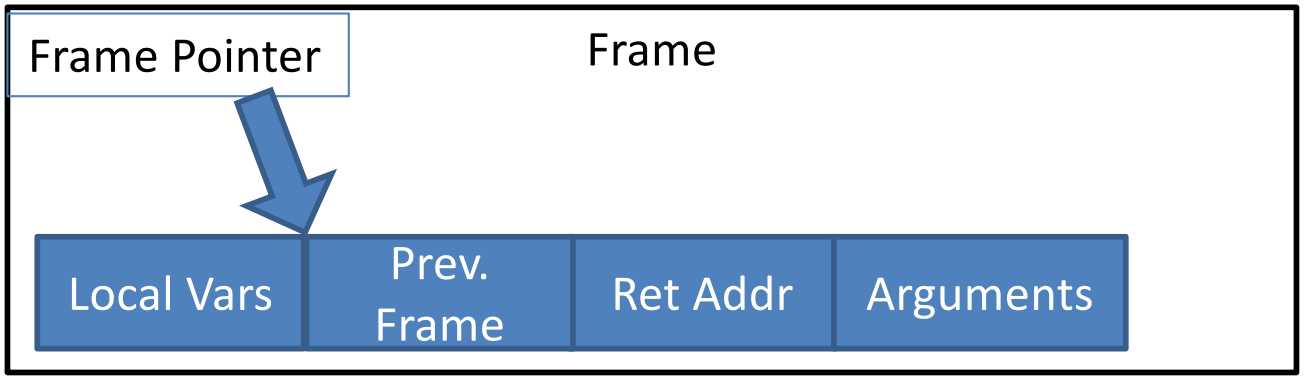
Low memory addresses

High memory addresses

```
main():
    call foo()
    call bar()
foo():
   f_glob = malloc(0x100)
   call bar()
bar()
   b_loc = 7;
```

Stack Frames

- Frame: local data for a function
  - Arguments
  - Return address
  - Previous frame
  - Local variables



Reminder: These are conventions

- Dictated by compiler
- Only instructions support by processor
  - Almost no structural notion of memory safety
    - Use of uninitialized memory
    - Use of freed memory
    - Memory leaks
- So how are they actually implemented?

Instruction Syntax

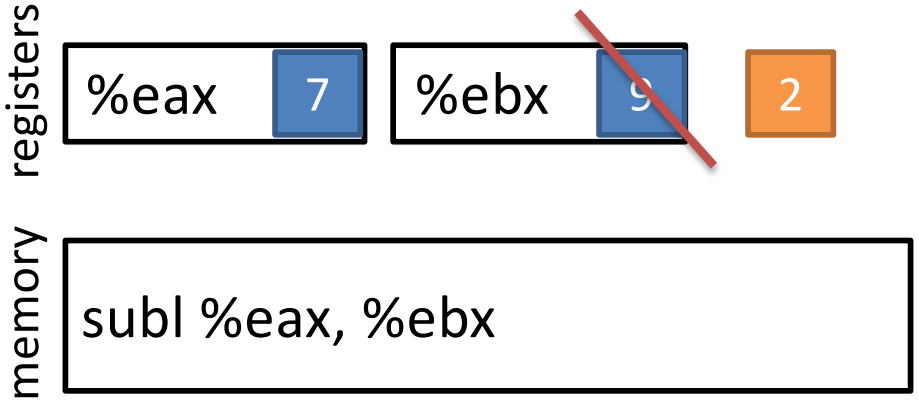
Examples:

```
subl $16, %ebx

movl (%eax), %ebx
```

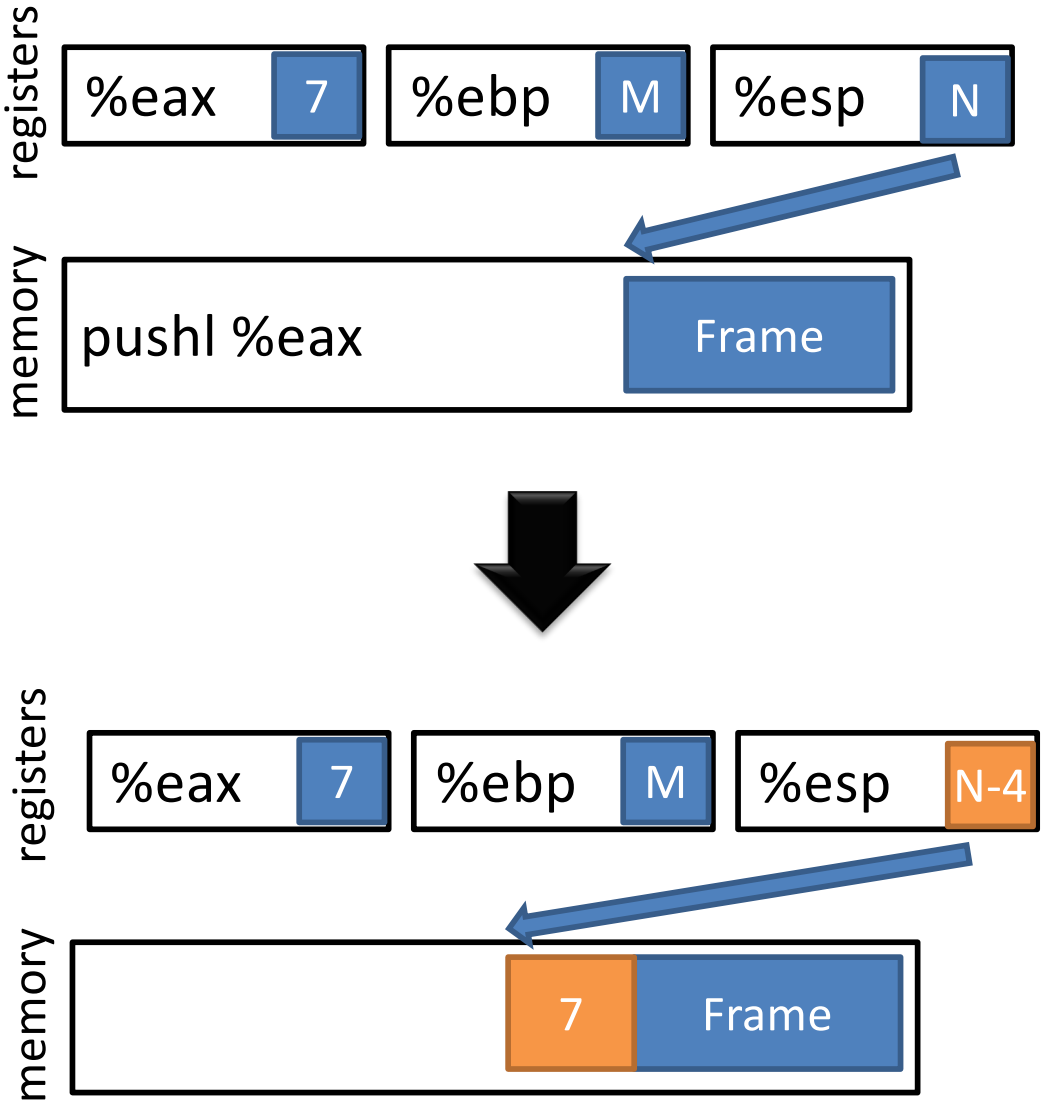
- Instruction ends with data length
- opcode, src, dst
- Constants preceded by $
- Registers preceded by %
- Indirection uses ( )

Register Instructions: sub



- Subtract from a register value

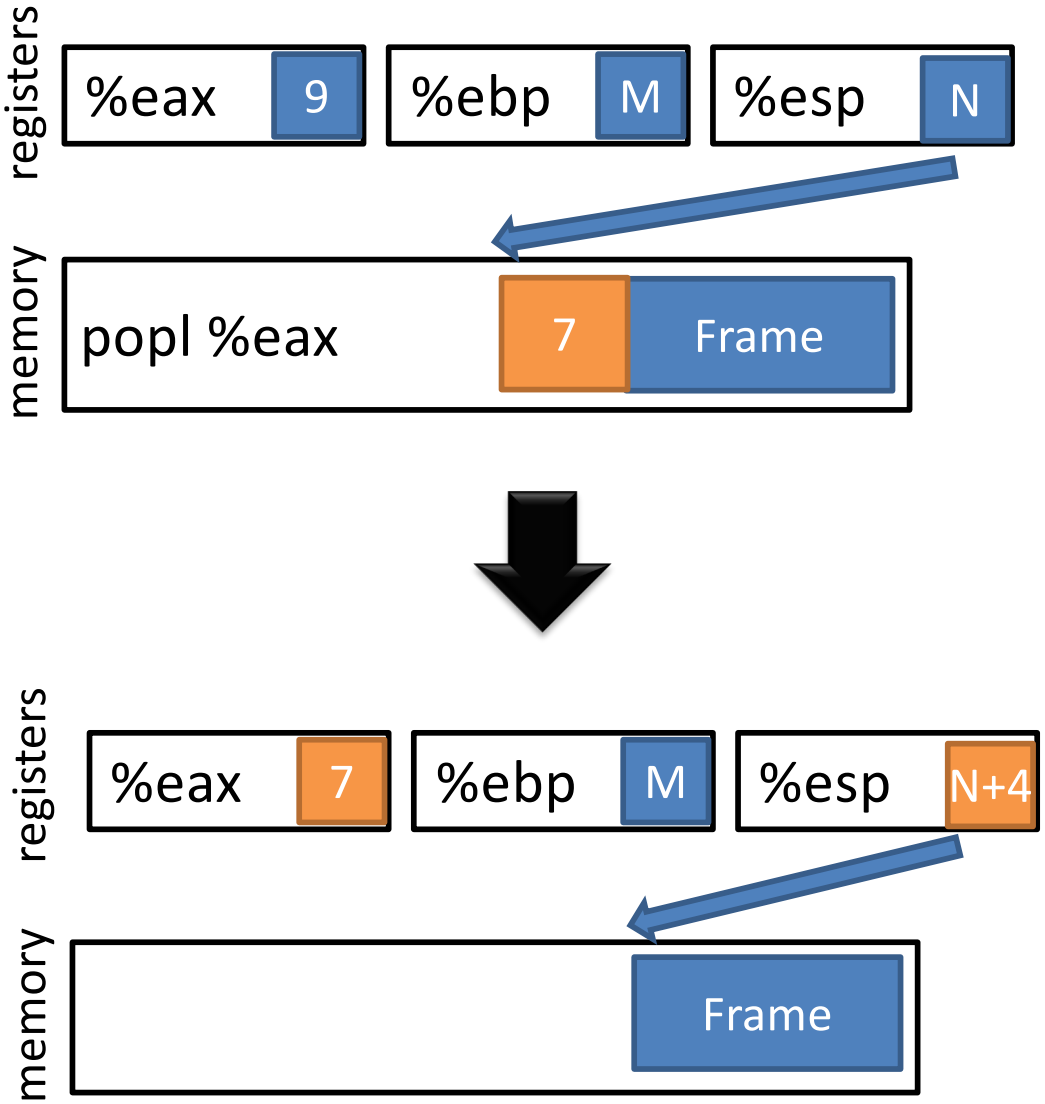
Frame Instructions: push



- Put a value on the stack
  - Pull from register
  - Value goes to %esp
  - Subtract from %esp
- Example:

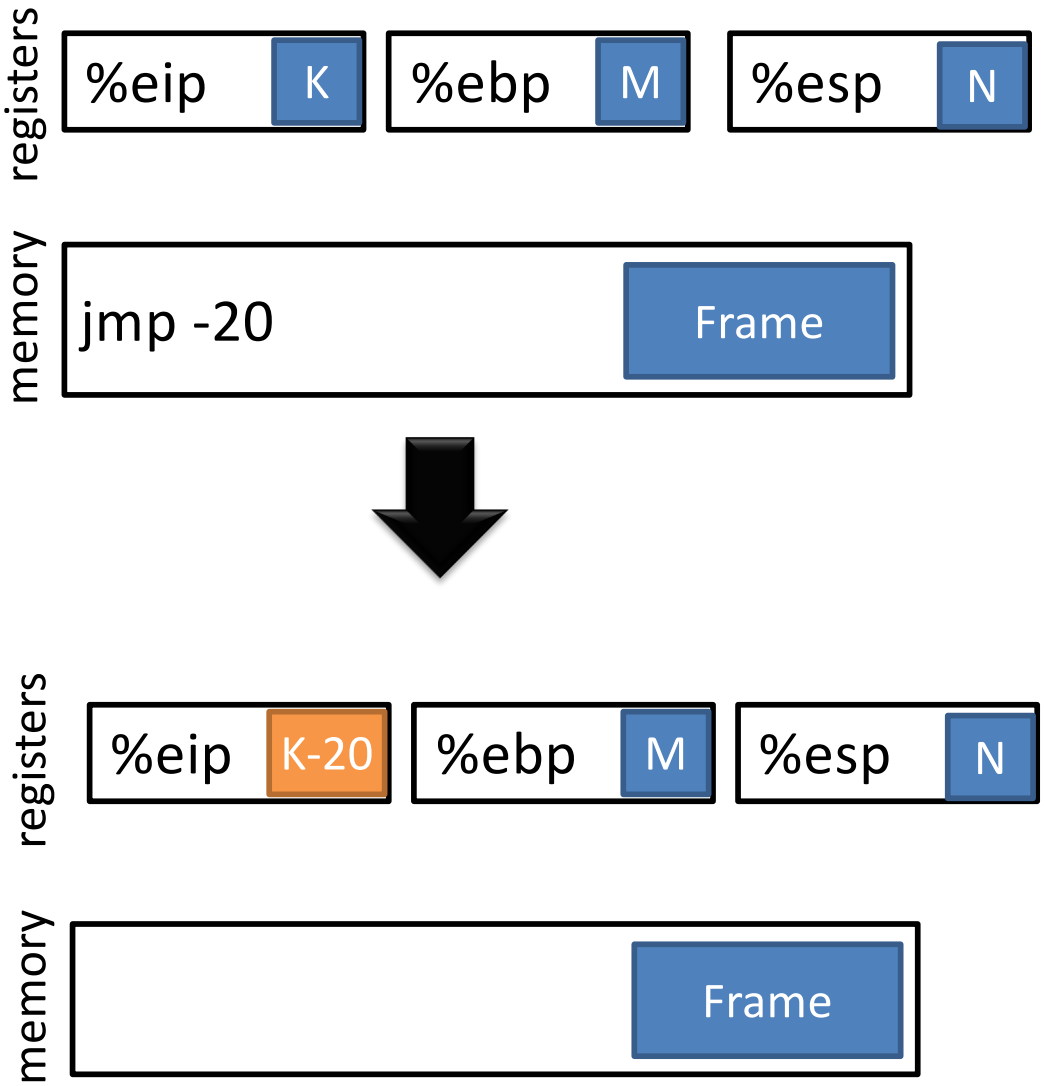
pushl %eax

Frame Instructions: pop



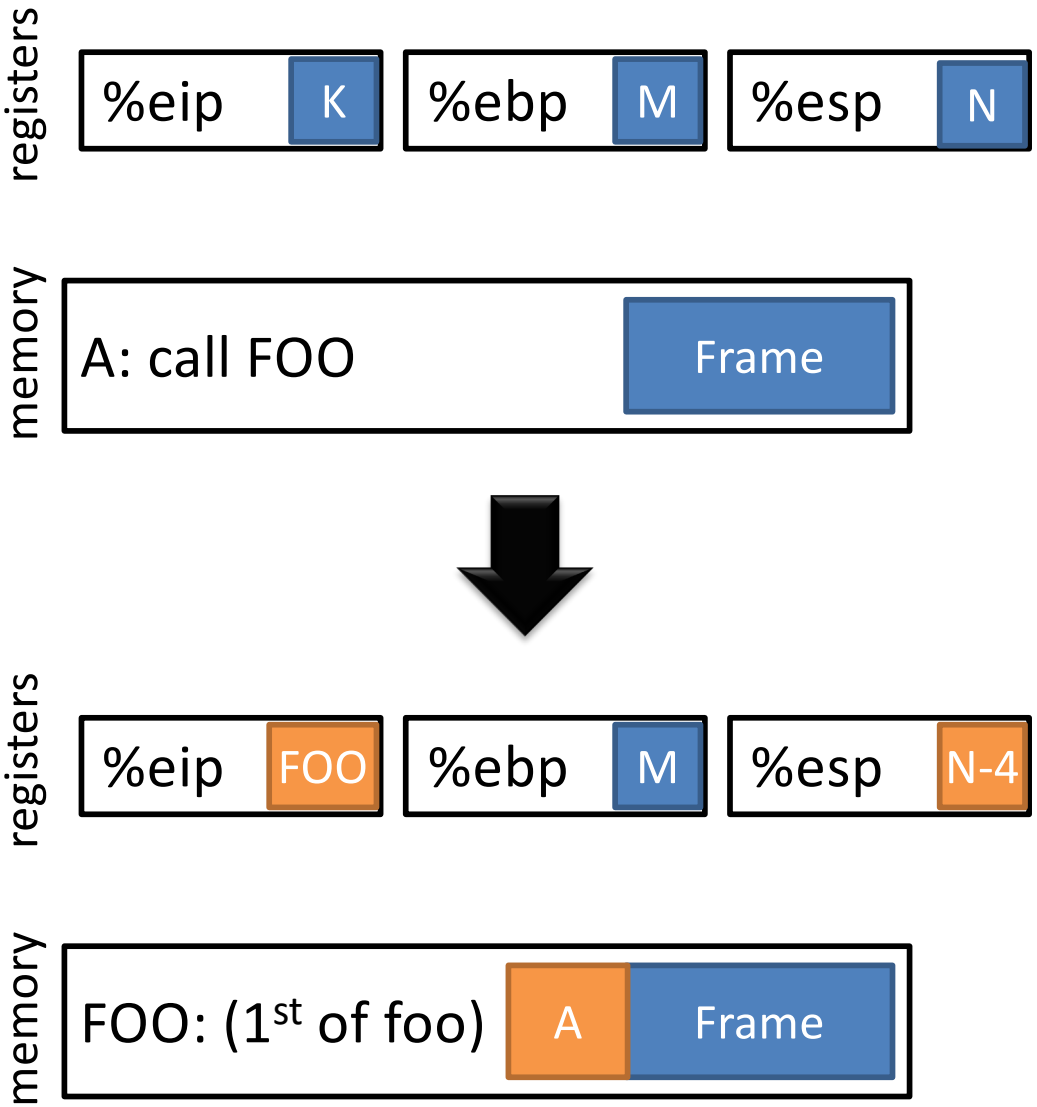
- Take a value from the stack
  - Pull from stack pointer
  - Value goes from %esp
  - Add to %esp

Control flow instructions: jmp



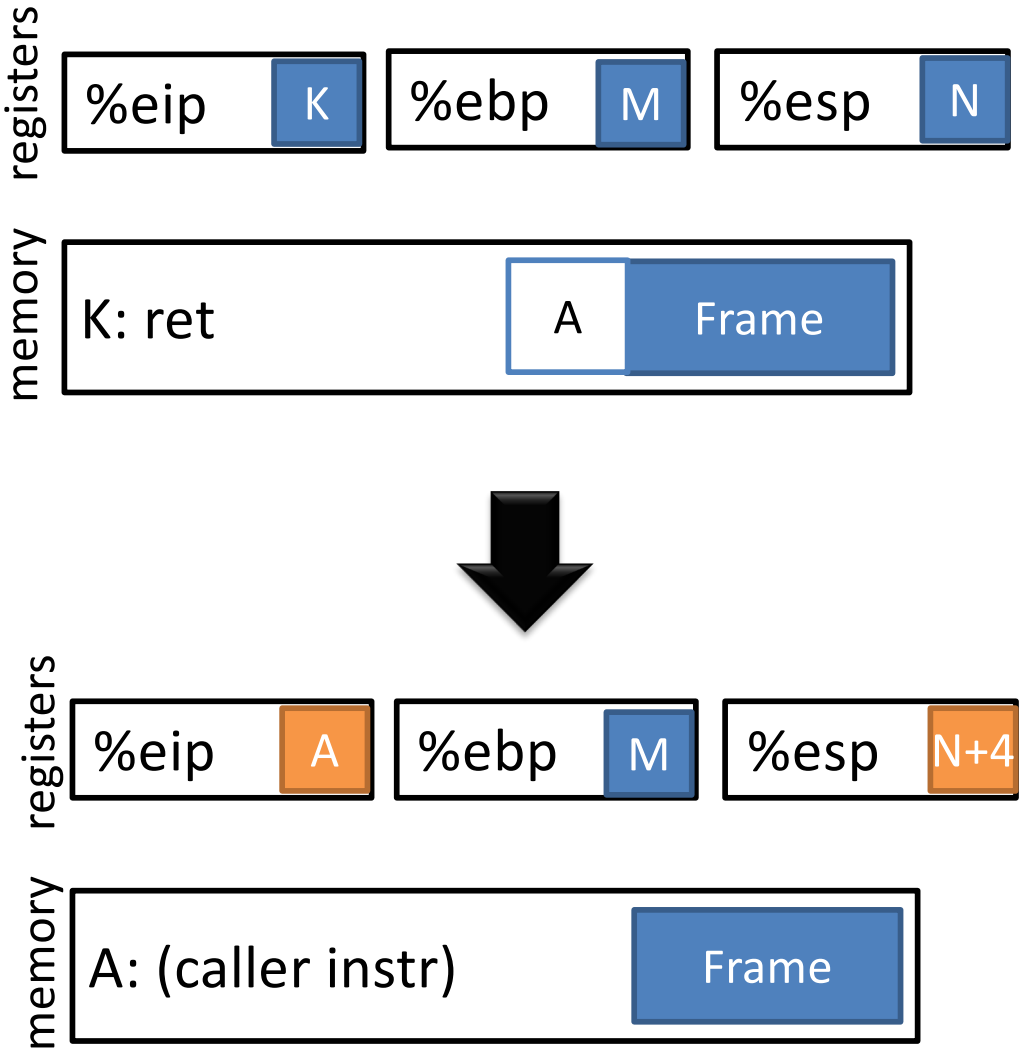
- %eip points to the currently executing instruction (in the text section)
- Has unconditional and conditional forms
- Uses relative addressing

Control flow instructions: call



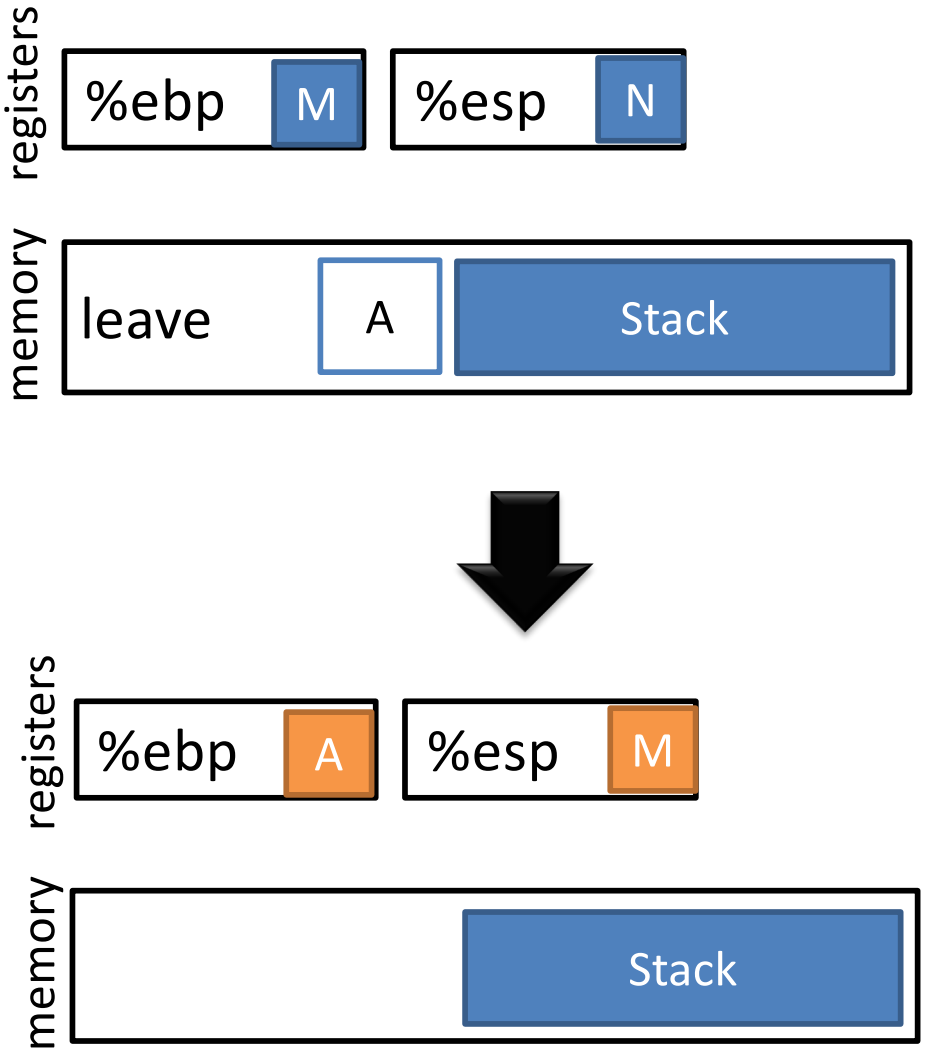
- Saves the current instruction pointer to the stack
- Jumps to the argument value

Control flow instructions: ret



- Pops the stack into the instruction pointer

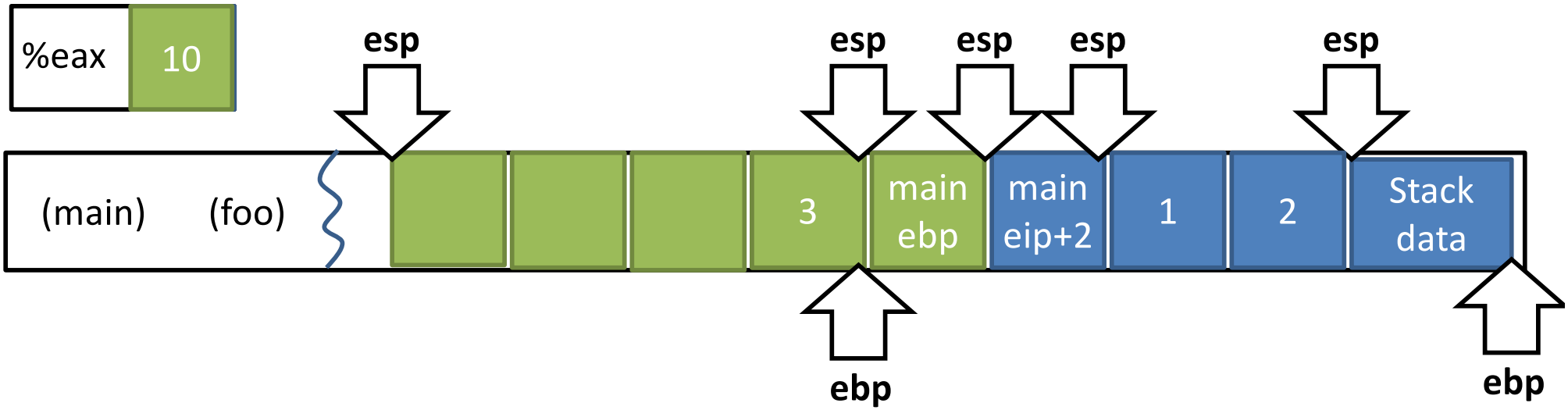
Stack instructions: leave



- Equivalent to

```
movl %ebp, %esp
popl %ebp
```

Implementing a function call




```
main:
        …
eip     subl    $8, %esp
eip     movl    $2, 4(%esp)
eip     movl    $1, (%esp)
eip     call    foo
eip     addl    $8, %esp
        …
```

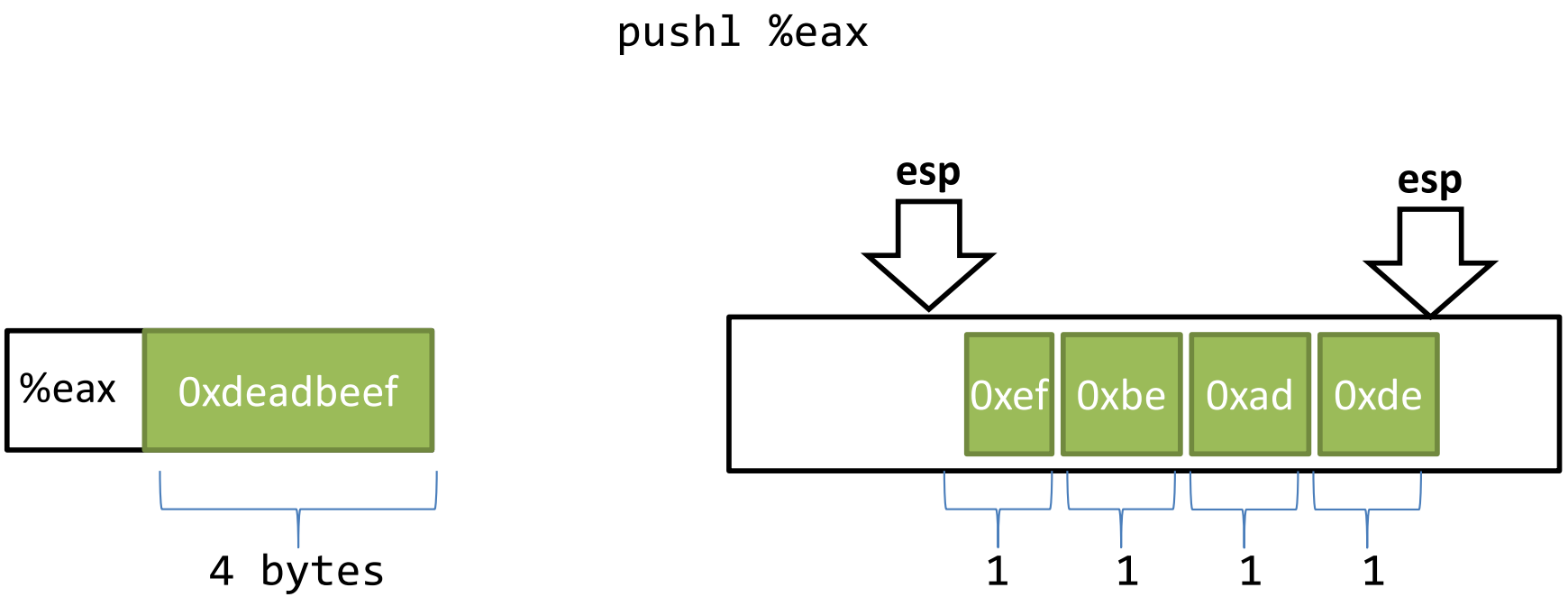
```
foo:
eip     pushl   %ebp
eip     movl    %esp, %ebp
eip     subl    $16, %esp
eip     movl    $3, -4(%ebp)
eip     movl    8(%ebp), %eax
eip     addl    $9, %eax
eip     leave
eip     ret
```

Function Calls: High level points

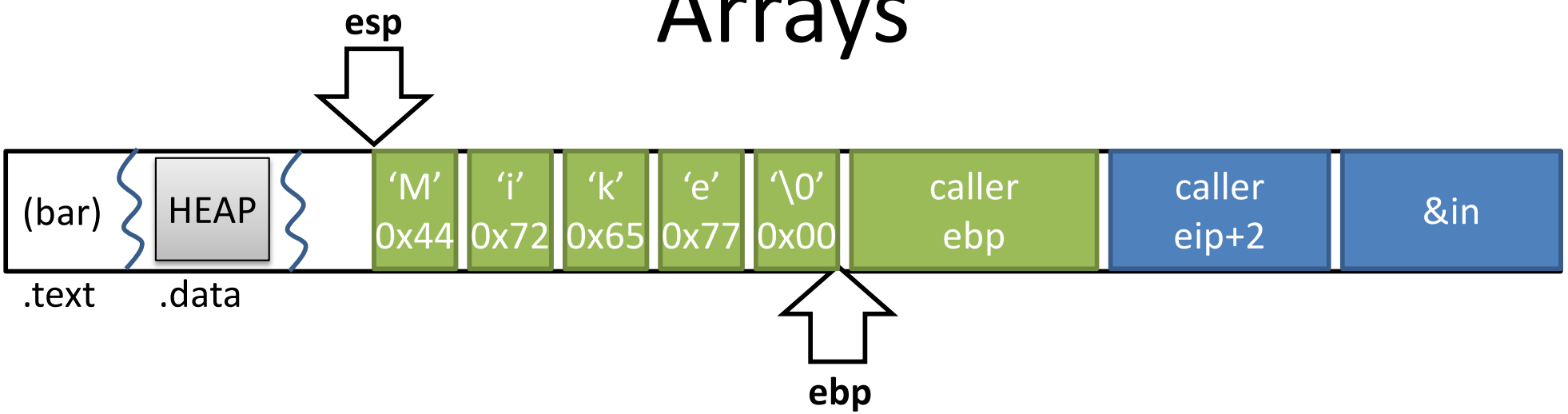
- Locals are organized into stack frames
  - Callees exist at lower address than the caller
- On call:
  - Save %eip so you can restore control
  - Save %ebp so you can restore data
- Implementation details are largely by convention
  - Somewhat codified by hardware

Data types / Endianness

- x86 is a little-endian architecture



Arrays




```
void bar(char * in){
  char name[5];
  strcpy(name, in);
}
```

```
bar:
  pushl  %ebp
  movl   %esp, %ebp
  subl   $5, %esp
  movl   8(%ebp), %eax
  movl   %eax, 4(%esp)
  leal   -5(%ebp), %eax
  movl   %eax, (%esp)
  call   strcpy
  leave
  ret
```

Next Time

Exploiting buffer overflows

Tools: GCC

```
gcc –O0 –S program.c –o program.S –m32
```

```
gcc –O0 –g program.c –o program –m32
```

Tools: GDB

```
gdb program
(gdb) run
(gdb) disassemble foo
(gdb) quit
```

Tools: objdump

```
objdump –Dwrt program
```

Tools: od

```
od -x program
```

Summary

- Basics of x86
  - Process layout
  - ISA details
  - Most of the instructions that you'll need
- Introduced the concept of a buffer overflow
- Some tools to play around with x86 assembly